Département d'informatique
Faculté des sciences
Université de Sherbrooke

Essai exploratoire sur l'utilisation du système de typage d'Eiffel dans la définition d'un nouveau langage de base de données relationnelle

Par
Maxime Routhier

Remis à
Luc Lavoie

Dans le cadre de l'activité pédagogique
IFT 855 Essai

Sherbrooke
15 mars 2021

Table des matières

Chapitre 1 : Introduction

Le présent essai s'inscrit dans le cadre de l'un des projets de recherche du Groupe de recherche interdisciplinaire en informatique de la santé (GRIIS) de l'Université de Sherbrooke, codirigé par le docteur Jean-François Ethier et le professeur Luc Lavoie. Le projet de recherche, dont il est question, constitue une tentative (peut-être ambitieuse) de définir un nouveau langage de base de données relationnelle en vue de remplacer SQL, le langage standard des bases de données relationnelles. Le nouveau langage, nommé Discipulus, bien que sa conception soit avancée, n'est pas encore complètement défini.

Objectifs de la recherche

Les objectifs du présent essai sont :

- contribuer à la définition du nouveau langage de base de données relationnelle du GRIIS ;
- étudier des systèmes de typage dans le but d'éclairer le GRIIS quant aux décisions à prendre concernant le système de typage du nouveau langage.

Problématique

Contexte général de la recherche

SQL souffre de nombreux défauts (Date, 2015a). D'une part, SQL est inutilement complexe. Pour citer C. J. Date, « [...] SQL is such a complex language, and provides so many different ways of doing the same thing, and is subject to so many exceptions and special cases [...]. » En effet, c'est l'une des raisons pour lesquelles la norme SQL elle-même est si difficile à comprendre. « [...] [The] official standard is not particularly easy to read. In places in fact, it is well-nigh impenetrable. » (Date & Darwen, 1997)

D'autre part, SQL présente de nombreux écarts par rapport à la théorie relationnelle. Pour citer Date encore une fois, « [...] SQL departs from relational theory in all too many ways; duplicate rows and nulls are two obvious examples, but they're not the only ones. As a consequence, the language gives you rope to hang yourself with, as it were. » Soit dit en passant, c'est l'une des raisons pour lesquelles les produits SQL ne parviennent pas à fournir une solution satisfaisante au problème de la mise à jour des vues. « I neither know nor care whether nonrelational views—by which I mean ones involving nonrelational concepts such as duplicate rows and nulls—are

updatable (though I'm pretty sure there's no way to update such views that can be defended as logically correct). » (Date, 2013)

Ensuite, au lieu d'utiliser SQL, de nombreux programmeurs préfèrent, pour diverses raisons :

- soit utiliser des outils tels que les ORM (*object-relational mapping tools*) pour accéder à une base de données relationnelle,
- soit abandonner complètement la technologie relationnelle et utiliser des technologies telles que les bases de données objet.

Pour ce qui est des ORM, la solution n'est pas satisfaisante (Neward, 2006). Pour citer Ted Neward : « […] *Object/Relational Mapping is the Vietnam of Computer Science*. It represents a quagmire which starts well, gets more complicated as time passes, and before long entraps its users in a commitment that has no clear demarcation point, no clear win conditions, and no clear exit strategy. »

Quant aux bases de données objet, la solution, même après des décennies de recherche, souffre toujours de plusieurs problèmes non résolus qui l'empêchent de remplacer la technologie relationnelle, comme espéré à l'origine (Nagy, 2007). Voici quelques raisons tirées de la référence (Nagy, 2007) :

- « lack of, or poor support for, a simple but expressive query language for writing ad hoc queries, if ad hoc queries are supported at all;
- inability, or difficulty, of sharing of data between applications, due to fact that persistent data is often tightly coupled with a particular programming language and paradigm;
- data manipulation is chiefly performed in a navigational ('pointer chasing') rather than a declarative (for example, relational algebra) manner, despite the fact that navigational databases (network and hierarchical) had been proven to be inferior, both in theory and practice, to declarative (relational) databases a long time ago [Date and Codd, 1975][.] »

Partant de ce fait, le GRIIS entreprend la définition d'un nouveau langage de base de données relationnelle : Discipulus.

Les objectifs généraux du langage sont :

1. Le langage doit permettre de décrire des modèles de données, ou, plus précisément, des schémas logiques de base de données. En particulier, la description de schémas physiques ou conceptuels dépasse le cadre du langage.

2. Dans un contexte de développement logiciel, le langage doit permettre de décrire des modèles du patron d'architecture Modèle-Vue-Contrôleur (MVC). En particulier, la description de vues ou de contrôleurs dépasse le cadre du langage.
3. Le langage doit réaliser le modèle relationnel de données tel que décrit par C. J. Date et Hugh Darwen. Plus précisément, le langage doit être conforme aux exigences énoncées dans le *Troisième Manifeste* (Date & Darwen, 2006). *Note :* Il peut exister des raisons valables d'ignorer une exigence particulière, mais toutes les implications doivent être comprises et soigneusement pesées avant de choisir un cours différent.
4. Le langage doit fournir un modèle de programmation propre et uniforme et doit tirer parti des meilleures capacités des bases de données et des langages de programmation pour, entre autres, le typage statique, l'optimisation et le développement modulaire.

Il est à noter que l'approche de Discipulus est en contraste avec celle qui tente d'unifier les bases de données et les langages de programmation (Atkinson & Buneman, 1987). Les concepteurs de Discipulus ne croient pas en un tel langage universel. Soit dit en passant, le problème d'unifier les bases de données et les langages de programmation, même après des décennies de recherche, n'est toujours pas résolu de manière satisfaisante. Pour citer William R. Cook et Ali H. Ibrahim, « The problem of integrating databases and programming languages has been open for nearly 45 years. During this time much progress has been made, in exploring specialized database programming languages, orthogonal persistence, object-oriented databases, transaction models, data access libraries, embedded queries, and object-relational mapping. While new solutions are proposed every year, none has yet proven fully satisfactory. » (Cook & Ibrahim, 2005)

Contexte spécifique de la recherche

Le GRIIS s'interroge sur le choix du système de typage du nouveau langage. Notamment, le GRIIS se questionne sur le choix du modèle d'héritage. Deux candidats pour le rôle ont été retenus dans le cadre des travaux antérieurs du GRIIS : le système de typage de Tutorial D et celui d'Eiffel. La décision n'est pas très facile à prendre. En relation avec ce problème de recherche, voir la référence (Abouaddaoui, 2012).

Les objectifs du langage en lien avec le système de typage sont :

5. Le langage doit être typé, fortement vérifié (*strongly checked*) et vérifié statiquement (*statically checked*). (Voir Glossaire.)
6. Le langage doit fournir des mécanismes d'abstraction procédurale (déclaration de procédures et fonctions), d'abstraction de données (déclaration de types abstraits) et d'abstraction un peu plus générale (modules).

7. Le langage doit prendre en charge le sous-typage et l'héritage. *Note :* Le sous-typage et l'héritage sont des concepts distincts. Pour citer John C. Mitchell, « Perhaps the most common confusion surrounding object-oriented languages is the difference between subtyping and inheritance. The simplest distinction between subtyping and inheritance is this: *Subtyping is a relation on interfaces, inheritance is a relation on implementations*[1]. » (Mitchell, 2002) (Voir Glossaire.)
8. Dans la mesure du possible, les types scalaires et non scalaires doivent être traités uniformément, notamment en ce qui a trait à l'abstraction de données, au sous-typage et à l'héritage. (Voir Glossaire.)

Structure de l'essai

Comme mentionné plus haut dans la section « Objectifs de la recherche », l'un des objectifs du présent essai est d'étudier des systèmes de typage dans le but d'éclairer le GRIIS quant aux décisions à prendre concernant le nouveau langage. Cela dit, cet essai concentre son attention entièrement sur le système de typage d'Eiffel (et non sur celui de Tutorial D). Ainsi, le corps de l'essai consiste essentiellement en un chapitre assez long sur ce dernier système de typage.

Le chapitre qui vient d'être mentionné explore l'utilisation du système de typage d'Eiffel dans un contexte de langage de base de données relationnelle. Les sujets suivants y sont traités :

- types référence,
- types tuple,
- types relation,
- opérateurs de tuple,
- opérateurs de relation,
- bases de données.

Enfin, cet essai suppose que le lecteur est minimalement aux faits :

- de la théorie des types en général ;
- du modèle relationnel ;
- du langage de programmation Eiffel.

[1] D'autres points de vue existent cependant dans la littérature; voir par exemple (Taivalsaari, 1996).

Chapitre 2 : Le système de typage d'Eiffel

L'objectif du présent chapitre est d'explorer l'utilisation du système de typage d'Eiffel dans un contexte de langage de base de données relationnelle.

Le langage de programmation Eiffel a été conçu au milieu des années 80 par Bertrand Meyer. C'est un langage typé, vérifié statiquement et orienté objet basé sur la classe (c*lass-based object-oriented*). (Voir Glossaire.) Le langage prend en charge l'héritage multiple et les classes paramétrées (généricité, polymorphisme paramétré borné). Il a une syntaxe de type Pascal, bien qu'il ne soit pas une extension de Pascal. Une caractéristique importante d'Eiffel est sa prise en charge de la programmation par contrat (*design by contract*), avec des fonctionnalités intégrées qui permettent de vérifier au moment de l'exécution les antécédents, les conséquents, les variants et les invariants. Les principaux travaux de Meyer qui ont conduit au développement d'Eiffel incluent ceux sur l'héritage multiple (Meyer, 1988) [2], la généricité (Meyer, 1986) [3] et la programmation par contrat (Meyer, 1992).

Eiffel est un langage orienté objet à usage général et non un langage de base de données relationnelle. Par conséquent, le système de typage d'Eiffel ne peut pas être utilisé tel quel dans la définition d'un nouveau langage de base de données. Le présent chapitre est donc une tentative de modifier le système de typage d'Eiffel pour l'adapter à un langage de base de données. En particulier, les modifications apportées doivent tenir compte des objectifs du langage énoncés dans le chapitre 1. À titre de référence, un résumé des modifications apportées au système de typage tout au long de ce chapitre se trouve dans l'annexe A du présent essai.

Avant de commencer, cependant, il convient de noter les points suivants. Tout d'abord, tout au long de ce chapitre, les références à Eiffel doivent être comprises comme faisant référence à la version normative de ce langage. Le document de référence est :

> Ecma International, 2006. *Standard ECMA-367 - Eiffel: Analysis, Design and Programming Language*. 2e éd. Genève, Suisse ; Ecma International.

---

[2] Dans cet article, Meyer explique pourquoi l'héritage multiple est (selon lui) essentiel à la bonne conception d'un logiciel.

[3] Dans cet article, Meyer explique pourquoi un langage orienté objet suffisamment expressif doit prendre en charge à la fois l'héritage et la généricité. De plus, Meyer y montre que l'héritage est le mécanisme le plus puissant : on peut simuler la généricité avec l'héritage, mais pas l'inverse.

Ensuite, ce chapitre utilise les termes et concepts de langage de programmation tels que définis en Eiffel. Ce qui suit passe en revue un certain nombre de ces termes et concepts :

- Un **type** est la description d'un ensemble de structures de données (objets), caractérisées par les opérations qui leur sont applicables et par les propriétés formelles de ces opérations. Chaque type est défini à partir d'une classe, y compris les types de base tels que *BOOLEAN*, *CHARACTER*, *INTEGER* et *REAL*.
- Une **classe** est une implémentation partielle ou totale d'un type (ou d'un *pattern* de type si la classe est générique).
- Le **sous-typage** (*conformance* dans la terminologie Eiffel) est une relation binaire réflexive et transitive sur les types qui satisfait la propriété de subsomption : si un type *T'* est un sous-type d'un type *T* (ou si un type *T* est un sur-type d'un type *T'*), alors chaque expression de type *T'* a également le type *T*. Le sous-typage implique que les valeurs d'un type sont substituables aux valeurs d'un autre. Plus précisément, si *T'* est un sous-type de *T*, alors un contexte qui attend une expression de type *T* peut prendre une expression de type *T'* sans introduire d'erreur de type.
- L'**héritage** est un mécanisme de réutilisation d'implémentations (classes) qui permet de définir une nouvelle classe à partir de celles existantes en ajoutant ou en adaptant leurs caractéristiques, auquel cas il est dit que la classe hérite d'autres classes. Si une classe *C'* hérite d'une classe *C*, *C'* est une sous-classe de *C* et *C* est une sur-classe de *C'*. (Dans la terminologie Eiffel : *C'* est un descendant de *C* et *C* est un ancêtre de *C'*.) En Eiffel, deux formes d'héritage sont disponibles : conformant et non conformant. L'héritage conformant introduit une relation de sous-typage entre les types basés sur les classes en question, tandis que l'héritage non conformant n'en introduit aucune.

Aussi, dans ce chapitre, les termes et concepts suivants sont utilisés :

- Chaque type est un sous-type de lui-même.
- Un sous-type d'un sous-type de *T* est un sous-type de *T*.
- Si *T'* est un sous-type de *T* et que *T'* et *T* sont distincts, alors *T'* est un sous-type *propre* de *T*.
- Chaque type est un sur-type de lui-même.
- Un sur-type d'un sur-type de *T* est un sur-type de *T*.
- Si *T* est un sur-type de *T'* et que *T* et *T'* sont distincts, alors *T* est un sur-type *propre* de *T'*.

Des remarques analogues s'appliquent aux sous-classes et sur-classes. Donc :

- Chaque classe est une sous-classe d'elle-même.

- Une sous-classe d'une sous-classe de *C* est une sous-classe de *C*.
- Si *C'* est une sous-classe de *C* et que *C'* et *C* sont distinctes, alors *C'* est une sous-classe *propre* de *C*.
- Chaque classe est une sur-classe d'elle-même.
- Une sur-classe d'une sur-classe de *C* est une sur-classe de *C*.
- Si *C* est une sur-classe de *C'* et que *C* et *C'* sont distinctes, alors *C* est une sur-classe *propre* de *C'*.

Types référence

En Eiffel, les types sont divisés en deux catégories : les types cohésifs (*expanded* dans la terminologie Eiffel) et les types référence. Les valeurs d'un type cohésif sont des objets, tandis que les valeurs d'un type référence sont des références (pointeurs) vers des objets. Les types cohésifs utilisent une sémantique de copie : un objet est copié pendant l'affectation. Les types références utilisent une sémantique de référence : l'affection produit deux références vers le même objet. La différence clé est que le partage d'objets est possible dans ce dernier cas, mais non dans le premier.

En raison de ce qui précède, Eiffel inclut deux opérateurs d'égalité : « = » et « ~ ». L'opérateur « ~ » dénote toujours l'égalité de valeur (le contenu des objets). L'opérateur « = » dénote l'égalité de valeur lorsqu'il est utilisé avec des types cohésifs, et l'égalité de référence lorsqu'il est utilisé avec des types références.

À l'inverse, le *Troisième Manifeste* interdit expressément les types pointeur : « No database relvar shall include an attribute of type pointer. » (Date & Darwen, 2006) Selon Date et Darwen, les pointeurs sont non nécessaires et ils ne font qu'ajouter de la complexité. Il est à noter que le *Troisième Manifeste* n'interdit pas aux *implémentations* d'utiliser des pointeurs. Ce qui importe, c'est que les pointeurs ne soient pas *exposés* dans le modèle. Les concepteurs de Discipulus sont entièrement d'accord avec ce qui précède et ne souhaitent certainement pas inclure les types pointeur (types référence) dans Discipulus.

Afin de rendre le système de typage d'Eiffel conforme à cette exigence, une modification possible consiste à simplement enlever les types référence et, par conséquent, à traiter tous les types comme cohésifs. Cependant, cette modification enlève en même temps le sous-typage. En effet, Eiffel ne permet pas à un type d'être un sous-type (propre) d'un type cohésif : « No type conforms directly to an expanded type. [...] [An] expanded type [...] still conforms, of course, to itself. » (Ecma International, 2006) Le but visé par cette restriction est de permettre aux implémentations d'Eiffel (compilateurs ou interpréteurs Eiffel) de représenter les valeurs cohésives sans utiliser de

pointeurs. (Sans cette restriction, les implémentations d'Eiffel doivent représenter les valeurs cohésives par des pointeurs afin de permettre à une variable de type cohésif *T* de se voir affecter une valeur d'un sous-type *T'* de *T* ayant plus d'attributs que *T*.) Par contre, puisque les concepteurs de Discipulus prévoient, pour des raisons dépassant le cadre de cet essai, représenter toute valeur (plus grande que la taille d'un pointeur) par un pointeur, cette restriction n'apporte plus d'avantages. Par conséquent, il possible de laisser tomber cette restriction et de conserver ainsi le sous-typage.

Cela nous amène à trois premières modifications au système de typage d'Eiffel :

1. Enlever les types références et, par conséquent, traiter tous les types comme cohésifs.
2. Enlever la contrainte de validité « No type conforms directly to an expanded type. » et, par conséquent, permettre le sous-typage entre les types cohésifs.
3. Enlever les opérateurs d'égalité « ~ » et « /~ ». Avec la Modification 1, les opérateurs « = » et « /= » ont la même sémantique que les opérateurs « ~ » et « /~ », respectivement.

Il est intéressant de noter que la Modification 1 enlève en même temps les *nulls* (valeurs *void* dans la terminologie Eiffel).

Types tuple

Eiffel prend en charge les types tuple. Cela dit, comme nous le verrons dans un instant, la notion de tuple d'Eiffel n'est pas exactement celle du modèle relationnel.

Comme nous l'avons mentionné plus haut, Eiffel est un langage basé sur la classe. Les systèmes basés sur la classe sont assez *centrés* sur les classes. Pour ajouter un nouveau type d'objet au système, une classe décrivant les propriétés de ce type d'objet doit d'abord être définie. Dans le cas d'Eiffel, cette restriction est quelque peu relâchée. En effet, Eiffel prend en charge (depuis Eiffel 4) une notion de classe anonyme (tuple).

Les types tuple d'Eiffel peuvent être considérés comme des types obtenus à partir de classes anonymes réduites. Par exemple, le type *TUPLE* [*a* : *X* ; *b* : *Y* ; *c* : *Z*] agit comme le type obtenu à partir d'une classe (sans nom) ayant trois attributs *a*, *b* et *c* (de types *X*, *Y* et *Z*, respectivement) et trois procédures d'affectation (*setters*) correspondantes. Les étiquettes peuvent être omises : le type *TUPLE* [*X*, *Y*, *Z*] décrit des séquences de valeurs dont la première doit être de type *X*, la seconde de type *Y* et la troisième de type *Z*. La seule différence est la perte de la notation $t.x_i$ (*t.a*, *t.b* et *t.c* dans l'exemple précédent) pour accéder aux composants d'un tuple *t* (bien qu'il soit toujours possible d'utiliser *t.item*(*i*) qui retourne un *ANY*), et de même pour la modification des composants (il est toujours possible d'utiliser *t.put*(*x*, *i*)). Étant anonymes, les classes sous-

jacentes aux types tuple ne nécessitent aucune déclaration de classe spécifique ; elles sont implicitement définies par les types tuple[4].

Les valeurs des types tuple peuvent être construites à l'aide du constructeur « [] »[5]. Par exemple, l'expression [*True*, *3 + 4*, "*Ce texte*"] dénote un tuple de type *TUPLE* [*BOOLEAN*, *INTEGER*, *STRING*]. En Eiffel, cette dernière expression est connue sous le nom de tuple manifeste (*invocation de sélecteur de tuple* dans la terminologie du *Troisième Manifeste*). Il est à noter que les étiquettes ne sont pas permises dans les tuples manifestes (en Eiffel).

Un type tuple peut être un sous-type d'un autre type tuple. Par exemple, le type *TUPLE* [*X*, *Y*] est un sous-type du type *TUPLE* [*X*] ; le type *TUPLE* [*X*] est un sous-type du type *TUPLE*. Cette règle de sous-typage est appelée « sous-typage de largeur » (*width subtyping*) (pour emprunter un terme à la théorie des types (Pierce, 2002)). Il est important de noter que, en Eiffel, le sous-typage des types tuple ignore les étiquettes : le type *TUPLE* [*a* : *X* ; *b* : *Y*] est un sous-type du type *TUPLE* [*c* : *X*]. En fait, *les étiquettes ne jouent aucun rôle dans la sémantique des tuples* (en Eiffel). Eiffel inclut également le « sous-typage de profondeur » (*depth subtyping*) : le type *TUPLE* [*X*] est un sous-type du type *TUPLE* [*Y*] ssi *X* est un sous-type de *Y* (encore une fois, indépendamment de la présence d'étiquettes). Remarquons que le type *TUPLE* est un sur-type de tout type tuple ; le type *TUPLE* [*ANY*] est un sur-type de tout type tuple d'au moins un élément ; et ainsi de suite. (*ANY* est le type maximal en Eiffel.)

Comme nous venons de le voir, la notion de tuple d'Eiffel est basée sur la notion de tuple en mathématiques : un tuple est une collection ordonnée d'éléments de domaines potentiellement distincts. (Eiffel permet les étiquettes, mais leur seule utilisation est de permettre l'accès basé sur le nom aux composants de tuple, garantissant statiquement le type du résultat.) Il existe plusieurs différences logiques importantes entre cette dernière notion de tuple et celle de son homologue du modèle relationnel. Le tableau suivant résume ces différences.

---

[4] Plus précisément, chaque type tuple est basé (directement ou indirectement) sur une même classe fictive *TUPLE*. La classe est définie dans la bibliothèque EiffelBase et fournit les fonctionnalités applicables à tous les tuples. Il est important de noter qu'Eiffel ne permet pas de déclarer une classe qui hérite de la classe *TUPLE*.

[5] Dans la terminologie du *Troisième Manifeste* : les valeurs des types tuple peuvent être *sélectionnées* (ou spécifiées) à l'aide du *sélecteur de tuple* « [] ».

Tableau 1 : Synthèse des différences entre la notion de tuple du *Troisième Manifeste* et celle d'Eiffel

	Troisième Manifeste	Eiffel
Ordre des composants	non ordonnés	ordonnés de gauche à droite
Étiquettes (identifiants d'attribut)	obligatoires ; font partie du type	optionnelles ; ne font pas partie du type
Sous-typage	sous-typage de profondeur uniquement	sous-typage de largeur et de profondeur

En raison de ce qui précède, les types tuple d'Eiffel violent de nombreuses exigences du *Troisième Manifeste*. En ce qui concerne l'ordre des composants de tuple et les étiquettes, les concepteurs de Discipulus ne désirent certainement pas s'écarter du modèle relationnel de la sorte, et commettre ainsi les mêmes erreurs que SQL. (Les colonnes des tables SQL sont ordonnées de gauche à droite et peuvent être anonymes ; pour une discussion détaillée des conséquences indésirables de cet état de fait, voir (Date, 2006)) De là, il y a (au moins) deux possibilités pour rendre le système de typage d'Eiffel conforme aux exigences en lien avec l'ordre des composants et les étiquettes. Ou bien les types tuple relationnels sont introduits (avec une syntaxe différente) dans le système ; par conséquent, deux notions de tuple différentes coexisteront au sein du même système. Ou bien les types tuple sont modifiés pour les rendre relationnels. Dans le but de garder Discipulus aussi simple que possible (du moins pour le moment), cette dernière option (rendre les tuples d'Eiffel relationnels) a été choisie dans le présent essai.

Pour ce qui est du sous-typage, le *Troisième Manifeste* est assez strict sur le sujet : « If D supports type inheritance, then such support shall conform to the inheritance model defined in Part IV of [the *Manifesto*] book[6]. » (Date & Darwen, 2006) (La version à jour du modèle d'héritage se trouve dans la référence (Date, 2016)) Cependant, une des raisons pour lesquelles le GRIIS se questionne sur le choix du système de typage de Discipulus est que le modèle de sous-typage/héritage du *Troisième Manifeste* ne semble pas satisfaisant (pour des raisons dépassant le cadre de cet essai). D'un autre côté, les règles de sous-typage des types tuple d'Eiffel (sous-typage de largeur et de profondeur) uniformisent davantage le traitement des types scalaires et non scalaires dans le système de typage d'Eiffel (un système orienté objet). En effet, dans un système orienté objet, un sous-type *T'* de *T* peut être utilisé dans n'importe quel contexte où *T* est attendu et, en particulier,

[6] Le modèle d'héritage du *Troisième Manifeste* est plutôt un modèle de sous-typage; il ne capture pas l'essence de l'héritage : un mécanisme de modification incrémentielle.

T' peut contenir plus d'information (plus d'attributs) que T (cette dernière information supplémentaire pouvant être « oubliée » sans danger dans un contexte où *T* est attendu). Pour ces raisons, il a été choisi dans cet essai de conserver le sous-typage de largeur des types tuple d'Eiffel et, par conséquent, d'ignorer les exigences du *Troisième Manifeste* en lien avec le sous-typage.

Cela nous amène à quatre autres modifications au système de typage d'Eiffel :

4. Rendre obligatoires les étiquettes de composant de tuple dans les « expressions de type » (*TUPLE* [...]) et dans les tuples manifestes.
5. Faire en sorte que les étiquettes fassent partie du type des tuples. Par exemple, le type *TUPLE* [*a* : *X*] n'est plus équivalent au type *TUPLE* [*b* : *X*].
6. Enlever la notion d'ordre pour les composants de tuple. Par exemple, le type *TUPLE* [*a*: *X* ; *b* : *Y*] est maintenant équivalent au type *TUPLE* [*b* : *Y* ; *a* : *X*].
7. Ajuster le sous-typage des types tuple pour tenir compte des Modifications 4, 5 et 6. Par exemple, le type *TUPLE* [*a* : *X* ; *b* : *Y*] est maintenant un sous-type du type *TUPLE* [*b* : *Y*] ; il n'est plus, par contre, un sous-type du type *TUPLE* [*c* : *X*].

Il faut noter que le choix de rendre les tuples d'Eiffel relationnels défait certaines fonctionnalités basées sur la notion de tuple d'origine (par exemple, le mécanisme d'agent). Cependant, ces fonctionnalités sont probablement récupérables par le fait que les tuples mathématiques ne sont que des tuples relationnels avec des étiquettes numériques[7].

Types relation

Eiffel ne prend (malheureusement) pas en charge les types relation. Cependant, considérant le fait qu'une relation est essentiellement un ensemble de tuples du même type (ou, compte tenu du sous-typage, de types étant des sous-types du même sur-type), il possible d'introduire les types relation dans le système avec la déclaration de classe suivante.

```
class
  RELATION [G -> TUPLE]
inherit
  LINKED_SET [G]
create
  make
feature
  ...
end
```

[7] Un tuple mathématique (v_1, v_2, ..., v_n) peut être vu comme un tuple relationnel {$1=v_1$, $2=v_2$, ..., $n=v_n$}.

La classe *RELATION* précédente est une classe générique avec une contrainte de généricité restreignant les types correspondants, les paramètres génériques effectifs (dans la terminologie Eiffel), à des sous-types du type *TUPLE*. La classe hérite de *LINKED_SET*, une classe générique de la bibliothèque EiffelBase implémentant la notion générale d'ensemble. *Note :* Il est important de comprendre ici que la préoccupation principale du présent essai concerne spécifiquement le modèle du système de typage et non son implémentation. Par conséquent, la mise en œuvre de la classe *RELATION* (par exemple, *LINKED_SET*) n'a pas vraiment d'importance ici. Ce qui compte, c'est la sémantique des relations (dans notre cas, l'interface de la classe *RELATION*). Hériter de *LINKED_SET* est toutefois utile pour explorer les idées mises de l'avant à l'aide d'un compilateur Eiffel existant. Pour le reste de ce chapitre, supposons que la classe *RELATION* est définie par le système (*built in*).

La classe *RELATION* décrit (comme toute classe générique) un constructeur de type (générateur de type dans la terminologie du *Troisième Manifeste*) ; pour dériver un type relation, un sous-type du type *TUPLE* doit être fourni. Par exemple, le type *RELATION* [*TUPLE* [*a* : *X* ; *b* : *Y* ; *c* : *Z*]] décrit des ensembles de tuples possédant au moins trois composants étiquetés *a*, *b* et *c* (de types étant des sous-types de *X*, *Y* et *Z*, respectivement).

Un type relation peut être un sous-type d'un autre type relation. Par exemple, le type *RELATION* [*TUPLE* [*a* : *X* ; *b* : *Y*]] est un sous-type du type *RELATION* [*TUPLE* [*b* : *Y*]]. Cela est dû à la règle de sous-typage pour les types dérivés génériquement : *A* [*X*] est un sous-type de *B* [*Y*] ssi *A* est un sous-type de *B* et *X* est un sous-type de *Y*. (Le cas où *A* est différent de *B* sera discuté ci-après.) Remarquons que le type *RELATION* [*TUPLE*] est un sur-type de tout type relation ; le type *RELATION* [*TUPLE* [*a* : *ANY*]] est un sur-type de tout type relation dont les tuples ont au moins un composant étiqueté *a* ; et ainsi de suite.

Il existe un autre moyen d'obtenir une relation de sous-type sur les types relation : l'héritage (plus précisément, l'héritage conformant ; Eiffel prend également en charge l'héritage non conformant, une forme d'héritage restreinte qui n'introduit pas de relation de sous-type). Prenons par exemple la déclaration de classe suivante.

```
class
  RELATION_1 [G -> TUPLE]
inherit
  RELATION [G]
create
  make
feature
  ...
end
```

RELATION_1 est une classe générique qui hérite de *RELATION*. La classe a la même contrainte de généricité (*TUPLE*) que la classe *RELATION*. (Il est à noter ici que *G*, jouant un rôle de paramètre générique effectif dans la clause « **inherit** *RELATION* [*G*] », doit obligatoirement être un sous-type de *TUPLE* ; la contrainte de généricité de la classe *RELATION_1* assure cette condition.) Prenons par exemple le type *RELATION_1* [*TUPLE* [*a* : *X* ; *b* : *Y*]]. Ce dernier est (sans surprise) un sous-type du type *RELATION* [*TUPLE* [*a* : *X*]]. Par contre, il *n'est pas* un sous-type du type *RELATION* [*TUPLE* [*a* : *X* ; *b* : *Y* ; *c* : *Z*]] (même si *RELATION_1* est une sous-classe de *RELATION*). Cela est dû à la règle de sous-typage, mentionnée plus tôt, pour les types dérivés génériquement. Remarquons que peu importe les types tuple fournis, un type relation obtenu à partir de *RELATION* ne sera jamais un sous-type d'un type relation obtenu à partir de *RELATION_1*. *Note :* La classe *RELATION_1* n'est pas très intéressante : elle n'apporte rien de plus que la classe *RELATION*. Son introduction ici vise plutôt à montrer le cas « minimal » d'héritage. (Des exemples plus intéressants seront donnés plus loin dans cette section.)

Supposons que la classe *RELATION_1* est définie par l'utilisateur. En Eiffel, une classe peut déclarer des attributs (constantes ou variables), des routines (procédures ou fonctions) et des invariants de classe. Considérons d'abord le cas des attributs, qui décrivent des champs qui se retrouvent dans toutes les instances de la classe. Est-il sensé de pouvoir définir des valeurs de « relation » qui contiennent des valeurs en dehors de leur ensemble de tuples ? Le présent essai répond à cette question par la négative. D'une part, ces « relations » ne sont pas des relations, par définition. D'autre part, ces « relations », lorsque rendues persistantes dans une base de données, violent le *Principe de l'Information* énoncé dans le *Troisième Manifeste* : « The entire information content of the database is represented in one and only one way: namely, as explicit values in attribute positions in tuples in relations. » (Date & Darwen, 2006) Pour cette raison, il a été choisi dans cet essai d'interdire tout moyen où une classe relation pourrait se retrouver avec un ou plusieurs attributs (à ne pas confondre avec les attributs du modèle relationnel !). Notamment, la déclaration d'attribut est interdite à l'intérieur d'une sous-classe de *RELATION*.

Considérons maintenant le cas des fonctions et des procédures. (En Eiffel, une fonction est une routine qui retourne un résultat, tandis qu'une procédure est une routine qui n'en retourne pas.)[8] La question intéressante ici n'est pas de savoir si Discipulus doit prendre en charge les déclarations de routine. Il a déjà été mentionné au chapitre 1 que le langage doit fournir des

[8] Il faut noter qu'une fonction (en Eiffel) permet de modifier l'état. Cependant, cela sera interdit dans Discipulus (c'est d'ailleurs le cas dans le *Troisième Manifeste*, où une fonction est appelée un opérateur en lecture seule).

mécanismes d'abstraction procédurale ; le *Troisième Manifeste* exige également la même chose : le langage doit permettre aux utilisateurs de définir et de détruire leurs propres fonctions (scalaires ou non scalaires) et procédures. La question intéressante est plutôt de savoir si les routines sont regroupées avec les types (plus précisément, si la définition d'une routine donnée fait partie de la définition d'une certaine classe). En Eiffel, toute routine appartient à une classe et, par voie de conséquence, a accès aux attributs et routines internes (encapsulés) de cette classe. Inversement, le *Troisième Manifeste* suggère fortement que les définitions de routine ne soient pas « intégrées » aux définitions de type. En ce qui concerne l'encapsulation, Date et Darwen soutiennent qu'il est préférable d'utiliser le mécanisme de sécurité (généralement inclus dans les systèmes de base de données) pour contrôler quelles routines sont autorisées à accéder aux caractéristiques internes de quels types. Aussi, pour citer Date et Darwen, « [The] idea of an operator being bundled in with some type works just fine so long as the operator in question takes exactly one operand [...]. But as soon as it takes two or more [...] a degree of arbitrariness, artificiality, asymmetry, and awkwardness inevitably creeps in. » (Date & Darwen, 2006) En substance, la réponse à la question précédente revient à choisir entre une organisation fonctionnelle (ou procédurale) de programme et une organisation orientée objet de programme. Les deux ont des avantages et des inconvénients (Mitchell, 2002). Le présent essai ne fournit pas de réponse ici. (Des recherches supplémentaires pourraient être requises.) Dans le but d'apporter des modifications au système de typage d'Eiffel uniquement lorsque cela est jugé nécessaire, il a été décidé dans cet essai de conserver l'approche orientée objet d'Eiffel. Notamment, la déclaration de routine est permise à l'intérieur d'une sous-classe de *RELATION*.

Enfin, examinons le cas des invariants de classe. Un invariant de classe est une assertion (expression booléenne) que toute instance de la classe doit satisfaire à chaque instant auquel l'instance est observable de l'extérieur. Observons maintenant que cette notion est similaire à, mais (comme nous le verrons dans un instant) pas tout à fait la même que, la notion de contrainte d'intégrité du *Troisième Manifeste*. Les contraintes d'intégrité du *Troisième Manifeste* se répartissent en deux grandes catégories : les contraintes de type et les contraintes de base de données. En bref, une contrainte de type définit l'ensemble de valeurs qui constituent un type donné, tandis qu'une contrainte de base de données limite l'ensemble de valeurs qu'une base de données peut prendre. Le tableau suivant résume les différences entre les contraintes d'intégrité du *Troisième Manifeste* et les invariants de classe d'Eiffel.

Tableau 2 : Synthèse des différences entre les contraintes d'intégrité du *Troisième Manifeste* et les invariants de classe d'Eiffel

	Troisième Manifeste		Eiffel
	Contrainte de type	Contrainte de base de données	Invariant de classe
Fait partie de	définition de type (scalaire)	définition de base de données (en l'absence de structure de schéma en Tutorial D)[9]	définition de classe
Peut référencer	composants de représentation du type en question uniquement	relvars (variables de relation) de base de données uniquement	tout attribut de la classe en question ainsi que tout objet partagé accessible[10]
Peut utiliser	toute fonction accessible depuis la portée incluant la définition de la contrainte	toute fonction accessible depuis la portée incluant la définition de la contrainte	toute fonction accessible depuis la portée incluant la définition de l'invariant
Vérifié lorsque	lors de l'exécution d'un opérateur de sélection pour le type en question	à la fin de chaque instruction (du moins conceptuellement)	au début et à la fin de chaque appel utilisant comme cible un objet de la classe en question[11]

Maintenant, le *Troisième Manifeste* exige que le langage fournisse des mécanismes pour définir et détruire des contraintes de type et des contraintes de base de données. Ignorons les contraintes de base de données pour le moment et concentrons-nous spécifiquement sur les contraintes de type. (La discussion des contraintes de base de données est reportée à la section « Bases de données » plus loin.) Dans cet ordre d'idées, remarquons deux choses. Tout d'abord, les contraintes de type du *Troisième Manifeste* ne peuvent faire partie que des définitions de type scalaire (puisque le *Troisième Manifeste* ne prend pas en charge les définitions de type

[9] Certaines contraintes de base de données (clés candidates) peuvent faire partie d'une déclaration de relvar. Cependant, ces contraintes peuvent être considérées comme du sucre syntaxique pour des contraintes (généralement plus longues) déclarées séparément.
[10] En Eiffel, il est possible de partager un objet en utilisant le mécanisme de *once function*.
[11] Plusieurs détails sont omis ici : (1) l'appel peut être autant un appel d'attribut qu'un appel de routine; (2) l'objet cible doit être une instance directe de la classe en question; (3) l'appel doit être qualifié (de la forme *a.r*(…) avec une cible explicite, ici *a*); (4) la surveillance d'invariant de classe doit être activée.

tuple/relation). Ainsi, les seules « contraintes de type » qui s'appliquent à un type tuple/relation sont celles impliquées par celles des types des composants de tuple. Deuxièmement, puisque le système de typage d'Eiffel modifié ne permet pas le partage d'objets (grâce à la Modification 1), les invariants de classe sont essentiellement équivalents aux contraintes de type. Par contre, le champ d'application des invariants de classe est moins restreint que celui des contraintes de type. En effet, il est possible (à moins d'en imposer spécifiquement la restriction) de déclarer des invariants dans des sous-classes de *RELATION* et, par conséquent, de définir des contraintes de type pour des types relation. Cette possibilité est-elle sensée ? Le présent essai répond positivement à cette question. Premièrement, elle uniformise davantage le traitement des types scalaires et non scalaires. Deuxièmement, l'interdire limite inutilement le système de typage. Troisièmement, elle ne viole aucune exigence du *Troisième Manifeste*. Pour ces raisons, il a été décidé dans cet essai de conserver cette possibilité. Par conséquent, la déclaration d'invariant est permise à l'intérieur d'une sous-classe de *RELATION*.

Après avoir analysé ce qui peut ou non être déclaré à l'intérieur d'une classe relation, revenons sur le sujet de l'héritage, sur comment une telle classe elle-même peut être déclarée. Commençons par le cas de l'héritage simple. La classe *RELATION_1* montrée précédemment n'est pas très intéressante (principalement par rapport à ce qu'elle apporte de plus que la classe *RELATION*) : les routines et invariants déclarés à l'intérieur de *RELATION_1* (tout comme *RELATION*) ne peuvent supposer la présence d'aucun composant de tuple particulier (en parlant un peu vaguement). Considérons les deux déclarations de classe suivantes.

```
class
  RELATION_2 [G -> TUPLE [a : BOOLEAN; b : INTEGER]]
inherit
  RELATION [G]
create
  make
feature
  ...
end
```

```
class
  RELATION_3
inherit
  RELATION [TUPLE [a : BOOLEAN; b : INTEGER]]
create
  make
feature
  ...
end
```

Les deux classes héritent de *RELATION* et toutes deux restreignent le type des tuples à des sous-types de *TUPLE* [*a* : *BOOLEAN*; *b* : *INTEGER*]. D'un côté, la classe *RELATION_2* est générique et est presque identique à *RELATION_1* ; la contrainte de généricité n'est que plus « restrictive ». D'un autre côté, la classe *RELATION_3* est non générique et hérite d'un type relation fixe. Quelle est la différence pratique entre les deux classes ? Le tableau suivant résume les différences à l'aide d'exemples de code.

Tableau 3 : Synthèse des différences pratiques entre *RELATION_2* et *RELATION_3*

	RELATION_2	*RELATION_3*
Obtenir un type relation	*RELATION_2* [*TUPLE* [*a* : *BOOLEAN*; *b* : *INTEGER*]]	*RELATION_3*
« Extension » par dérivation générique	*RELATION_2* [*TUPLE* [*a* : *BOOLEAN*; *b* : *INTEGER*; *c* : *STRING*]]	impossible
« Extension » par héritage	**class** *RELATION_X* [*G* -> *TUPLE* [*a* : *BOOLEAN*; *b* : *INTEGER*; *c* : *STRING*]] **inherit** *RELATION_2* [*G*] …	impossible

En ce qui concerne l'obtention d'un type relation, *RELATION_3* permet une syntaxe plus courte que *RELATION_2*. En contrepartie, *RELATION_3* ajoute un niveau d'indirection : pour vérifier la présence d'un composant de tuple, les utilisateurs doivent aller à la déclaration de la classe. Quant à « l'ajout de composants de tuple », *RELATION_2* est plus extensible (ou réutilisable) que *RELATION_3*. En effet, *RELATION_3*, contrairement à *RELATION_2*, fixe une fois pour tout le type (déclaré) des tuples[12]. En somme, les deux approches présentent des avantages et des inconvénients. Les deux approches doivent-elles être prises en charge par Discipulus ? Sinon, quelle approche choisir ? Idéalement, une seule approche devrait être adoptée (du moins pour le moment) afin de garder Discipulus aussi simple que possible. Ensuite, il semble que *RELATION_2* est conceptuellement beaucoup plus proche de *RELATION* (entre autres les deux sont des classes génériques) et que ses avantages l'emportent sur ceux de *RELATION_3*

[12] Il faut noter qu'il est toujours possible de déclarer un invariant dans une sous-classe de *RELATION_3* garantissant que tous les tuples sont d'un sous-type propre particulier du type tuple déclaré. Cependant, les invariants de classe (contrairement aux contraintes de généricité) ne font pas partie du système de typage statique. Par conséquent, un compilateur peut ne pas être en mesure de garantir cette dernière assertion qui devra alors être vérifiée à l'exécution.

(principalement par rapport à la réutilisabilité). Pour ces raisons, il a été décidé dans le présent essai d'interdire l'approche *RELATION_3*.

Passons au cas de l'héritage multiple. Soit *RELATION_4* la classe relation (similaire à *RELATION_2*) suivante.

```
class
  RELATION_4 [G -> TUPLE [c : STRING]]
inherit
  RELATION [G]
create
  make
feature
  ...
end
```

Observons maintenant les deux déclarations de classe suivantes.

```
class
  RELATION_5 [G -> TUPLE [a : BOOLEAN; b : INTEGER; c : STRING]]
inherit
  RELATION_2 [G]
  RELATION_4 [G]
create
  make
feature
  ...
end
```

```
class
  RELATION_6 [G -> TUPLE [a : BOOLEAN; b : INTEGER]]
inherit
  RELATION_2 [G]
  EXCEPTIONS
    undefine
      is_equal, copy
    end
create
  make
feature
  ...
end
```

RELATION_5 hérite à la fois de *RELATION_2* et *RELATION_4*. La règle en jeu (satisfaite ici) est que la contrainte de généricité de *RELATION_5* doit être à la fois un sous-type de celle de *RELATION_2* et de celle de *RELATION_4*. Quant à *RELATION_6*, elle hérite à la fois de *RELATION_2* et *EXCEPTIONS*. *EXCEPTIONS* est une classe de la bibliothèque EiffelBase qui

fournit diverses fonctionnalités pour contrôler la gestion des exceptions[13]. *RELATION_6* illustre l'un des rôles de l'héritage dans la méthode Eiffel : en tant que mécanisme d'importation de modules. Ici, *RELATION_6* n'hérite d' *EXCEPTIONS* que pour utiliser ses routines (et non pour définir une spécialisation du type *EXCEPTIONS*). Ces deux classes relation (*RELATION_5* et *RELATION_6*) sont-elles sensées ? Pour ce qui est de *RELATION_5*, le présent essai répond positivement à cette question. Le cas d'héritage multiple de *RELATION_5* peut en fait être considéré comme une généralisation de celui d'héritage simple de *RELATION_2* discuté précédemment. Quant à *RELATION_6*, la décision n'est pas aussi simple. À première vue, il peut être tentant d'interdire les situations dans lesquelles une classe relation hérite d'une classe non-relation, en particulier pour appliquer l'exigence mentionnée plus tôt à l'effet qu'une classe relation ne doit pas se retrouver avec des attributs (encore une fois, à ne pas confondre avec les attributs du modèle relationnel !). Cependant, cela entraverait la réutilisabilité. Dans la méthode Eiffel, le mécanisme d'importation de modules est l'héritage[14]. Ainsi, une classe non-relation intéressante uniquement en tant que module (plus précisément, encapsulant un certain nombre de routines et n'ayant pas attribut, comme c'est le cas avec *EXCEPTIONS*) ne pourrait pas être « importée » par une classe relation. D'un autre côté, permettre à une classe relation d'hériter d'une classe non-relation (à condition que cette dernière n'ait pas d'attribut) peut être vu comme allant à l'encontre du principe de l'accès uniforme d'Eiffel, stipulant que, pour les clients d'une classe (autres classes qui utilisent ses fonctionnalités), un attribut est indiscernable d'une fonction sans arguments. En effet, l'utilisateur devra savoir si les requêtes sans arguments d'un module sont des fonctions ou des attributs pour déterminer si une classe relation peut en hériter. Ce fait pourrait donner lieu à des difficultés car la documentation client standard d'une classe est conçue de manière à ne pas révéler si une caractéristique donnée est un attribut ou une fonction. Mais réflexion faite, il semble que l'avantage de la réutilisabilité l'emporte sur celui de l'accès uniforme. Pour cette raison, il a été choisi dans cet essai de permettre le cas de *RELATION_6* (plus précisément, de permettre à une classe relation d'hériter d'une classe non-relation à condition que cette dernière n'ait pas d'attribut).

Cela nous amène à deux autres modifications au système de typage d'Eiffel :

8. Prédéfinir la classe *RELATION* comme suit.

---

[13] En particulier, *EXCEPTIONS* fournit une fonction *exception* donnant le code de la dernière exception, et une procédure *raise* qui déclenchera explicitement une « exception de développeur » avec un code qui peut ensuite être détecté et traité.

[14] Eiffel n'introduit pas de notion de module au-dessus des classes. En d'autres termes, les classes sont la seule forme de module en Eiffel.

```
class
  RELATION [G -> TUPLE]
feature
  ...
end
```

Note : Un compilateur Eiffel n'a pas besoin de construire la classe explicitement ; mais la façon dont il gère les types relation doit être la même que si la classe était réellement présente.

9. Définir que toute classe qui est une sous-classe de *RELATION* appartient à la catégorie des classes relation. Soit *CR* une telle classe relation. Faire en sorte que *CR* soit valide seulement si *CR* est la classe *RELATION* ou bien *CR* a toutes les propriétés suivantes :
    a. *CR* est une classe générique avec exactement un paramètre générique formel.
    b. *CR* hérite d'au moins une classe relation.
    c. Soit *G* le paramètre générique formel de *CR*. Le paramètre générique effectif de toute classe relation héritée est *G*.
    d. Toute sur-classe propre (directe ou indirecte) de *CR* ne déclare aucun attribut.
    e. *CR* ne déclare aucun attribut.

Il est à noter que, dans la solution adoptée ici, les types relation sont en partie définis implicitement (par le mécanisme de généricité) et en partie définis explicitement (par le mécanisme d'héritage). Une autre approche (peut-être plus sensée) pourrait être de faire en sorte que les types relation soient entièrement définis implicitement (par exemple, en interdisant d'hériter de la classe *RELATION*) afin d'uniformiser le traitement des types relation et des types tuple, ces derniers étant définis implicitement. (À cette fin d'uniformité, une autre approche pourrait être de permettre d'hériter de la classe *TUPLE*.)

Opérateurs de tuple

Le *Troisième Manifeste* exige que le langage prenne en charge les opérateurs de tuple suivants :

- des opérateurs analogues aux opérateurs de renommage, de projection, d'extension et de jointure de l'algèbre relationnelle ;
- l'opérateur d'affectation « := » ;
- les opérateurs de comparaison « = » et « ≠ » ;
- un opérateur de sélection de tuple ;
- un opérateur d'extraction d'attribut (pour extraire une valeur d'un attribut spécifié à partir d'un tuple spécifié) ;
- des opérateurs pour effectuer une « imbrication » et « déimbrication » de tuple.

Le modèle d'héritage du *Troisième Manifeste* exige également que le langage prenne en charge les opérateurs suivants :

- un opérateur de *downcasting* qui inclut un test de type à l'exécution (le test est pour garantir la sûreté du typage) ;
- un opérateur de test de type (un opérateur booléen pour tester si une valeur spécifiée est d'un type spécifié).

Comme noté dans la section « Types tuple », Eiffel fournit le sélecteur de tuple « [] » et l'extracteur d'attribut « . » (la notation pointée). Avec les Modifications 4 à 7, le sélecteur et l'extracteur ont les propriétés requises par le *Troisième Manifeste*.

Eiffel fournit également l'opérateur d'affectation « := » pour affecter des valeurs de tuple à des variables de tuple. (Eiffel fournit l'opérateur d'affectation pour chaque type.) Rappelons que dans une affectation *v* := *x* (où *v* est une variable de tuple et *x* est une expression de tuple ; la remarque s'applique toutefois pour tout type), le type déclaré de *x* peut être un sous-type propre du type déclaré de *v*.

Dans la section « Types référence », nous avons vu qu'Eiffel fournit l'opérateur d'égalité « = » (et « /= »). Avec la Modification 1, l'opérateur « = » a la même sémantique que celle prescrite par le *Troisième Manifeste* (l'égalité de valeur). Cependant, bien que la sémantique soit équivalente, il reste une différence : à savoir la règle de typage. D'un côté, la fonction *is_equal* d'Eiffel est déclarée dans la classe *TUPLE* comme suit. (L'opérateur « = » repose sur la fonction *is_equal*.)[15]

```
is_equal (other : like Current) : BOOLEAN
  ...
```

La déclaration **like** *Current* dénote un type basé sur la classe courante (avec les mêmes paramètres génériques ou composants de tuple le cas échéant). Ainsi, par exemple, le fragment de code suivant est invalide.

[15] Soit dit en passant, la fonction *is_equal* (déclarée dans *ANY*) peut être redéfinie localement dans n'importe quelle classe pour tenir compte d'une notion d'égalité d'objet définie par le programmeur et adaptée à la sémantique spécifique de la classe. Par conséquent, si le programmeur ne fait attention, l'opérateur « = » d'Eiffel peut violer la propriété suivante requise par le *Troisième Manifeste* : « [If] (a) there exists [a determinate] operator *Op* (other than "=" itself) with a parameter *P* [...] such that (b) two successful invocations of *Op* that are identical in all respects except that the argument corresponding to *P* is *v1* in one invocation and *v2* in the other are distinguishable in their effect, then (c) *v1* = *v2* must evaluate to FALSE. » Des recherches supplémentaires pourraient être nécessaires.

```
t1 : TUPLE [a : BOOLEAN; b : INTEGER]
t2 : TUPLE [a : BOOLEAN]
...
t1 := [a : True; b : 3]
t2 := t1
print(t1.is_equal(t2))  -- warning: compile time type error!
```

L'expression *t1.is_equal*(*t2*) dans la dernière ligne ici échouera lors d'une vérification de type à la compilation (même si la valeur courante de *t1* aurait été égale à celle de *t2* au moment de l'exécution). La raison est que le type de *t2* n'est pas un sous-type de celui de *t1*. D'un autre côté, la règle de typage prescrite par le *Troisième Manifeste* est plus faible : soit la comparaison d'égalité *x* = *y* (où *x* et *y* sont des expressions) ; les types déclarés de *x* et *y* doivent se chevaucher. (Deux types *se chevauchent* si et seulement s'il existe au moins une valeur commune aux deux.) Ainsi, par exemple, l'expression *t1* = *t2* (où *t1* et *t2* sont les variables de l'exemple précédent) serait valide avec cette dernière règle de typage. Y a-t-il une règle de typage (parmi celles d'Eiffel et du *Troisième Manifeste*) plus sensée que l'autre ? À bien considérer les choses, les deux sont insatisfaisantes. D'un côté, celle d'Eiffel est désagréablement restrictive et ne suit pas la tradition mathématique ; *t2* = *t1* est valide alors que *t1* = *t2* ne l'est pas (où, encore une fois, *t1* et *t2* sont les variables de l'exemple précédent). D'un autre côté, la règle de typage du *Troisième Manifeste*, si elle est adoptée dans le système de typage d'Eiffel modifié, complexifie le langage. Le système de typage d'Eiffel ne prend pas en charge une notion de chevauchement de type. (La seule relation sur les types prise en charge par Eiffel est l'inclusion de type, c'est-à-dire le sous-typage.) Néanmoins, supposons pour le moment que le système de typage d'Eiffel modifié prenne en fait en charge une notion de chevauchement de type et que la règle de typage pour la fonction *is_equal* de la classe *TUPLE* soit celle du *Troisième Manifeste*. Examinons l'une des implications. Considérons le fragment de code suivant, où *X* n'est pas un sous-type de *Y* et vice versa.

```
t3 : TUPLE [a : X]
t4 : TUPLE [a : Y]
...
print(t3.is_equal(t4))  -- compile time type error?
```

Les types déclarés de *t3* et *t4* se chevauchent si et seulement s'il existe un type *Z* tel que *Z* est à la fois un sous-type de *X* et *Y*. Supposons pour simplifier que *X* et *Y* ne sont ni des types tuple, ni des types dérivés génériquement. Dans ce cas, l'expression *t3.is_equal*(*t4*) est valide si et seulement s'il existe une classe dans l'univers[16] qui hérite de *X* et *Y*. Cela peut être déterminé par l'analyse de

[16] En Eiffel, l'univers est l'ensemble des classes dans le système.

toutes les classes dans le système[17]. Cette complexité n'en vaut probablement pas la peine, surtout lorsqu'il y a une solution beaucoup plus simple : remplacer le type **like** *Current* du paramètre formel *other* (dans la déclaration de la fonction *is_equal* de la classe *TUPLE*) par le type *TUPLE*. Avec cette dernière solution, par exemple, l'expression *t3.is_equal*(*t4*) est valide indépendamment de l'existence ou non d'un type *Z* qui est à la fois un sous-type de *X* et *Y*. Si un tel *Z* n'existe pas, l'expression *t3.is_equal*(*t4*) est simplement équivalente à la constante booléenne *False* ; en particulier, l'expression ne lèvera pas d'erreur de type lors de l'exécution. Pour ces raisons, il a été décidé dans le présent essai d'adopter cette dernière solution.

Passons aux cas des opérateurs de *downcasting* et de test de type. Eiffel fournit une construction syntaxique appelée test d'objet. Un test d'objet, de la forme **attached** {*T*} *exp* **as** *x*, où *T* est un type, *exp* est une expression et *x* est un nom[18], est une expression booléenne ; sa valeur est vrai ssi *exp* est de type *T*. De plus, l'évaluation de l'expression a pour effet de laisser *x* dénoter la valeur de *exp* sur l'exécution d'une partie voisine du texte, appelée portée du test d'objet. Par exemple, dans l'instruction conditionnelle **if attached** {*T*} *exp* **as** *x* **then** *c1* **else** *c2* **end**, la portée du test d'objet est *c1*. Dans *c1*, *x* peut être utilisé comme une variable en lecture seule, sachant qu'elle est de type déclaré *T* et qu'elle a la valeur que *exp* avait lors de l'évaluation du test d'objet. La construction test d'objet permet d'obtenir à la fois la sémantique de l'opérateur de *downcasting* requis par le *Troisième Manifeste* et celle de l'opérateur de test de type, également requis par le *Troisième Manifeste*. Cependant, de manière analogue au paragraphe précédent, les règles de typage ne sont pas les mêmes : celles des opérateurs de *downcasting* et de test de type requièrent que les types déclarés de l'expression et du type (correspondant respectivement à *exp* et *T* dans la construction test d'objet) se chevauchent, tandis que celle du test d'objet n'a aucune restriction. Cela dit, pour les mêmes raisons exposées dans le paragraphe précédent, cet essai n'adoptera pas la règle du *Troisième Manifeste*.

Considérons maintenant les cas des opérateurs analogues aux opérateurs de renommage, de projection, d'extension et de jointure de l'algèbre relationnelle. Le *Troisième Manifeste* ne prescrit pas de règle de typage et de sémantique spécifiques pour ces opérateurs de tuple (ou leurs équivalents relationnels d'ailleurs). Le présent essai utilise donc les définitions d'opérateurs de tuple données dans le *Nouveau dictionnaire de base de données relationnelle* (Date, 2015b) :

---

[17] Des recherches supplémentaires pourraient être nécessaires pour déterminer l'impact sur le processus de compilation lorsque celui-ci est « local » et que l'univers n'est connu que lorsque le système entier est assemblé.

[18] Le nom *x* doit être différent de ceux de toutes les variables du contexte englobant.

- **renommage de tuple :** Soit *t* un tuple ayant un attribut appelé *a* et aucun attribut appelé *b*. Alors (et alors seulement) l'expression *t* RENAME {*a* AS *b*} dénote un renommage d'attribut sur *t*, et elle retourne le tuple qui est identique à *t* sauf que cet attribut *a* dans ce tuple est renommé *b*.
- **projection de tuple :** Soit *t* un tuple ayant des attributs appelés *a1*, *a2*, ..., *an* (et éventuellement d'autres). Alors l'expression *t* {*a1*, *a2*, ..., *an*} dénote la projection de *t* sur {*a1*, *a2*, ..., *an*}, et elle retourne le tuple obtenu en supprimant de *t* tous les composants autres que ceux correspondant aux attributs *a1*, *a2*, ..., *an*.
- **extension de tuple :** 1. (Première forme) Soit *t* un tuple n'ayant aucun attribut appelé *a*. Alors (et alors seulement) l'expression EXTEND *t* : {*a* := *exp*} retourne un tuple identique à *t* sauf qu'il a un attribut supplémentaire appelé *a*, avec une valeur qui est calculée en évaluant l'expression *exp* sur *t*. 2. (Deuxième forme) Soit *t* un tuple ayant un attribut appelé *a*. Alors (et alors seulement) l'expression EXTEND *t* : {*a* := *exp*} retourne un tuple identique à *t* sauf que la valeur de l'attribut *a* est remplacée par une valeur qui est calculée en évaluant l'expression *exp* sur *t*.
- **union de tuple/jointure de tuple :** Soit *t1* et *t2* des tuples tels que les attributs de même nom soient du même type et aient la même valeur. Alors (et alors seulement) l'expression *t1* UNION *t2* dénote l'union de *t1* et *t2*, et elle retourne le tuple qui est l'union de la théorie des ensembles de *t1* et *t2*.

Eiffel ne fournit (malheureusement) pas d'opérateurs similaires à ces derniers. De plus, les définitions du *Nouveau dictionnaire de base de données relationnelle* ne prennent pas en compte le modèle de sous-typage/héritage du *Troisième Manifeste* (sans parler du système de typage d'Eiffel). Le reste de cette section est donc une tentative de définir une règle de typage et une sémantique spécifique pour chacun de ces opérateurs de tuple, en tenant compte du système de typage d'Eiffel modifié. *Note :* Par souci de simplicité, la syntaxe pour appeler ces opérateurs sera basée sur celle donnée dans le *Nouveau dictionnaire de base de données relationnelle* (donc sur la syntaxe de Tutorial D).

Commençons par l'opérateur de projection de tuple. Cet essai propose la définition suivante.

> **projection de tuple :** Une projection de tuple, de la forme *tx* {*a1*, *a2*, ..., *an*}, où *tx* est une expression de tuple, *a1*, *a2*, ..., *an* sont des étiquettes du type déclaré de *tx* et $n \geq 0$, est une expression de tuple ; cette expression est évaluée à un tuple qui est dérivé de la valeur courante de *tx* en supprimant tous les composants autres que ceux correspondant aux étiquettes *a1*, *a2*, ..., *an*, et le type déclaré de cette expression est un type tuple qui est

dérivé du type déclaré de *tx* en supprimant tous les composants autres que ceux correspondant aux étiquettes *a1*, *a2*, ..., *an*.

Ensuite, pour l'opérateur de renommage de tuple, la définition suivante est proposée.

> **renommage de tuple :** Un renommage de tuple, de la forme *tx* **rename** {*a1* **as** *b1*, *a2* **as** *b2*, ..., *an* **as** *bn*}, où *tx* est une expression de tuple, *a1*, *a2*, ..., *an* sont des étiquettes distinctes du type déclaré de *tx*, *b1*, *b2*, ..., *bn* sont des étiquettes distinctes qui sont différentes de toutes celles du type déclaré de *tx* et $n \geq 0$, est une expression de tuple. Si la valeur courante de *tx* a une étiquette parmi *b1*, *b2*, ..., *bn*, alors une exception est levée ; sinon, cette expression est évaluée à un tuple qui est identique à la valeur courante de *tx* sauf que ces étiquettes *a1*, *a2*, ..., *an* dans ce tuple sont renommées *b1*, *b2*, ..., *bn*, respectivement, et le type déclaré de cette expression est un type tuple qui est identique au type déclaré de *tx* sauf que ces étiquettes *a1*, *a2*, ..., *an* dans ce type tuple sont renommées *b1*, *b2*, ..., *bn*, respectivement.

Puis, pour l'opérateur d'union de tuple (ou de jointure de tuple), la définition suivante est proposée.

> **union de tuple/jointure de tuple :** Une union de tuple (ou jointure de tuple), de la forme *tx1* **union** *tx2*, où *tx1* et *tx2* sont des expressions de tuple, est une expression de tuple. Soit *t1* et *t2* les valeurs courantes de *tx1* et *tx2*, respectivement, et soit *DT1* et *DT2* les types déclarés de *tx1* et *tx2*, respectivement. Si les composants de *t1* et *t2* de même étiquette n'ont pas la même valeur, alors une exception est levée ; sinon, cette expression est évaluée à un tuple qui est l'union de la théorie des ensembles de *t1* et *t2*, et le type déclaré de cette expression est un type tuple qui est l'union de la théorie des ensembles de *DT1* et *DT2*, en remplaçant par *ANY* le type de tous les composants de même étiquette n'étant pas du même type.

Enfin, pour l'opérateur d'extension de tuple, la définition suivante est proposée. *Note :* Par souci de simplicité, seul le cas où il n'y a qu'une seule affectation de composant est pris en compte. Les considérations impliquées dans le traitement de plusieurs affectations sont essentiellement simples.

> **extension de tuple :** Une extension de tuple, de la forme **extend** *tx* **as** *x* : {*a* := *exp*}, où *tx* est une expression de tuple, *x* est un nom[19], *a* est une étiquette et *exp* est une expression

[19] Le nom *x* doit être différent de ceux de toutes les variables du contexte englobant.

autorisée à contenir une référence à *x*[20], est une expression de tuple. Soit *t* la valeur courante de *tx*. Si *t* n'a aucun composant appelé *a*, alors cette expression est évaluée à un tuple qui est identique à *t* sauf qu'il y a un composant supplémentaire appelé *a*, avec une valeur qui est calculée en évaluant *exp* ; sinon, cette expression est évaluée à un tuple qui est identique à *t* sauf que la valeur du composant *a* est remplacée par une valeur qui est calculée en évaluant *exp*. Dans les deux cas, le type déclaré de cette expression est un type tuple qui est dérivé du type déclaré de *tx* en ajoutant (ou en remplaçant le composant *a* par) le composant « *a* : *DT* », où *DT* est le type déclaré de *exp*.

Les cas des opérateurs d'« imbrication » et de « déimbrication » de tuple ne sont pas abordés dans cet essai.

Cela nous amène à deux autres modifications au système de typage d'Eiffel :

10. Dans la déclaration de la fonction *is_equal* de la classe *TUPLE*, remplacer le type **like** *Current* du paramètre formel *other* par le type *TUPLE*.
11. Fournir les opérateurs de projection de tuple, de renommage de tuple, d'union/jointure de tuple et d'extension de tuple tels que définis dans la section « Opérateurs de tuple » du chapitre 2 du présent essai.

[20] Dans *exp*, *x* peut être utilisé comme une variable en lecture seule, sachant qu'elle est de type déclaré celui de *tx* et qu'elle a la valeur qui résulte de l'évaluation de *tx*.

Chapitre 3 : Conclusion

Comme mentionné au tout début dans l'introduction, le GRIIS s'interroge sur le choix du système de typage de Discipulus. Les deux candidats retenus pour le rôle sont le système de typage de Tutorial D et celui d'Eiffel. La décision n'est pas facile à prendre. Le présent essai ne fournit (malheureusement) pas de réponse définitive à cet égard. Par contre, cet essai peut être vu comme la réalisation d'une première étape fondamentale sur la voie d'une étude comparative complète des systèmes de typage de Tutorial D et d'Eiffel. Une fois terminée, cette étude comparative permettra de cerner les avantages et inconvénients de ces deux systèmes. Le GRIIS pourra alors prendre une décision éclairée sur le système de typage de Discipulus.

Faute de temps, deux sujets importants n'ont pas été abordés lors de l'investigation du système de typage d'Eiffel au chapitre 2, à savoir : les opérateurs de relation et l'intégration des variables de relation dans un « conteneur » appelé base de données. Toutefois, une première réflexion se dégage de l'essai :

> *Les définitions proposées pour les opérateurs de tuple analogues à ceux de renommage, de projection, d'extension et de jointure de l'algèbre relationnelle montrent que le système de typage d'Eiffel ne se prête pas naturellement aux règles de typage idéales pour ces opérateurs. Chaque règle de typage proposée est un cas particulier qui semble s'écarter du reste du système. La raison principale est que ces opérateurs, en plus de produire un tuple comme résultat, « construisent » un type tuple qui dépend de ceux des opérandes. Aussi, bien que le sujet des opérateurs de relation n'ait pas été abordé, il est raisonnable de supposer qu'en définissant les opérateurs de l'algèbre relationnelle, le défaut qui vient d'être mentionné ne fera que s'accentuer.*

Dans l'ensemble, cependant, il faut noter qu'aucun obstacle majeur n'a été rencontré lors de l'investigation du système de typage d'Eiffel. Ce dernier système peut, avec les modifications appropriées, être utilisé comme système de typage pour un langage de base de données relationnelle. Reste à savoir si l'intégration complète des technologies orientées objet et relationnelles fournie par un tel langage crée des avantages qui l'emportent sur la perte d'uniformité du modèle de programmation.

Annexe A : Modifications au système de typage d'Eiffel

À titre de référence, la présente annexe résume les modifications apportées au système de typage d'Eiffel tout au long du chapitre 2.

1. Enlever les types références et, par conséquent, traiter tous les types comme cohésifs.
2. Enlever la contrainte de validité « No type conforms directly to an expanded type. » et, par conséquent, permettre le sous-typage entre les types cohésifs.
3. Enlever les opérateurs d'égalité « ~ » et « /~ ». Avec la Modification 1, les opérateurs « = » et « /= » ont la même sémantique que les opérateurs « ~ » et « /~ », respectivement.
4. Rendre obligatoires les étiquettes de composant de tuple dans les « expressions de type » (*TUPLE* [...]) et dans les tuples manifestes.
5. Faire en sorte que les étiquettes fassent partie du type des tuples. Par exemple, le type *TUPLE* [*a* : *X*] n'est plus équivalent au type *TUPLE* [*b* : *X*].
6. Enlever la notion d'ordre pour les composants de tuple. Par exemple, le type *TUPLE* [*a*: *X* ; *b* : *Y*] est maintenant équivalent au type *TUPLE* [*b* : *Y* ; *a* : *X*].
7. Ajuster le sous-typage des types tuple pour tenir compte des Modifications 4, 5 et 6. Par exemple, le type *TUPLE* [*a* : *X* ; *b* : *Y*] est maintenant un sous-type du type *TUPLE* [*b* : *Y*] ; il n'est plus, par contre, un sous-type du type *TUPLE* [*c* : *X*].
8. Prédéfinir la classe *RELATION* comme suit.

```
class
  RELATION [G -> TUPLE]
feature
  ...
end
```

   Note : Un compilateur Eiffel n'a pas besoin de construire la classe explicitement ; mais la façon dont il gère les types relation doit être la même que si la classe était réellement présente.
9. Définir que toute classe qui est une sous-classe de *RELATION* appartient à la catégorie des classes relation. Soit *CR* une telle classe relation. Faire en sorte que *CR* soit valide seulement si *CR* est la classe *RELATION* ou bien *CR* a toutes les propriétés suivantes :
   a. *CR* est une classe générique avec exactement un paramètre générique formel.
   b. *CR* hérite d'au moins une classe relation.
   c. Soit *G* le paramètre générique formel de *CR*. Le paramètre générique effectif de toute classe relation héritée est *G*.

    d. Toute sur-classe propre (directe ou indirecte) de *CR* ne déclare aucun attribut.
    e. *CR* ne déclare aucun attribut.
10. Dans la déclaration de la fonction *is_equal* de la classe *TUPLE*, remplacer le type **like** *Current* du paramètre formel *other* par le type *TUPLE*.
11. Fournir les opérateurs de projection de tuple, de renommage de tuple, d'union/jointure de tuple et d'extension de tuple tels que définis dans la section « Opérateurs de tuple » du chapitre 2 du présent essai.

Glossaire

héritage : Un mécanisme de modification incrémentielle en présence d'une autoréférence liée tardivement (*late-bound self-reference*). (Taivalsaari, 1996)

langage basé sur la classe : Langage orienté objet qui repose sur des classes formant des modèles pour la génération de nouveaux objets. (Bruce, 2002)

langage fortement vérifié : Langage dans lequel aucune erreur interdite ne peut se produire lors de l'exécution (selon la définition d'erreur interdite). (Cardelli, 2004)

langage typé : Langage avec un système de typage (statique) associé, que les types fassent partie ou non de la syntaxe. (Cardelli, 2004)

langage vérifié statiquement : Langage dans lequel le bon comportement est déterminé avant l'exécution. (Cardelli, 2004)

non scalaire : Qui n'est pas scalaire ; c'est-à-dire ayant des composants visible par l'utilisateur. Les constructions non scalaires les plus importantes dans le modèle relationnel sont les tuples et les relations, où les « composants visibles par l'utilisateur » sont les attributs pertinents (et sans doute les tuples pertinents également, dans le cas d'une relation). (Date, 2015b)

polymorphisme : Capacité d'un fragment de programme à avoir plusieurs types (l'opposé du monomorphisme). (Cardelli, 2004)

polymorphisme paramétré : Polymorphisme dans lequel une fonction peut être appliquée à tous arguments dont les types correspondent à une expression de type impliquant des variables de type. (Mitchell, 2002)

polymorphisme par sous-typage : Polymorphisme dans lequel la relation de sous-type entre les types permet à une expression d'avoir plusieurs types possibles. (Mitchell, 2002)

scalaire : N'ayant pas de composants visibles par l'utilisateur. (Date, 2015b)

sous-typage : Une relation binaire réflexive et transitive sur les types. Elle satisfait la subsomption et affirme l'inclusion de collections de valeurs. (Cardelli, 2004)

subsomption : Une règle fondamentale du sous-typage, affirmant que si un terme a un type *A*, qui est un sous-type d'un type *B*, alors le terme a également le type *B*. (Cardelli, 2004)

système de typage : Une méthode syntaxique *tractable* pour prouver l'absence de certains comportements de programme en classant les phrases en fonction des sortes de valeurs qu'elles calculent. (Pierce, 2002)

type : Une collection de valeurs. Une estimation de la collection de valeurs qu'un fragment de programme peut prendre lors de l'exécution du programme. (Cardelli, 2004)

Bibliographie

Abouaddaoui, Z., 2012. *Contributions à la définition d'un nouveau langage d'exploitation des bases de données relationnelles,* s.l.: Mémoire de maîtrise, Université de Sherbrooke.

Atkinson, M. P. & Buneman, O. P., 1987. Types and Persistence in Database Programming Languages. *ACM Computing Surveys,* juin, 19(2), pp. 105-190.

Bruce, K. B., 2002. *Foundations of Object-Oriented Languages: Types and Semantics*. Cambridge, MA, États-Unis: The MIT Press.

Cardelli, L., 2004. Type Systems. Dans: A. B. Tucker, éd. *Computer Science Handbook*. 2e éd. Boca Raton, FL, États-Unis: Chapman & Hall/CRC.

Cook, W. R. & Ibrahim, A. H., 2005. *Integrating Programming Languages & Databases: What's the Problem?*. [En ligne]
Available at: https://www.cs.utexas.edu/~wcook/Drafts/2005/PLDBProblem.pdf

Date, C. J., 2006. A Sweet Disorder. Dans: *Date on Database: Writings 2000-2006*. New York, NY, États-Unis: Apress.

Date, C. J., 2013. *View Updating and Relational Theory: Solving the View Update Problem*. Sebastopol, CA, États-Unis: O'Reilly Media.

Date, C. J., 2015a. *SQL and Relational Theory: How to Write Accurate SQL Code*. 3e éd. Sebastopol, CA, États-Unis: O'Reilly Media.

Date, C. J., 2015b. *The New Relational Database Dictionary*. Sebastopol, CA, États-Unis: O'Reilly Media.

Date, C. J., 2016. *Type Inheritance and Relational Theory: Subtypes, Supertypes, and Substitutability*. Sebastopol, CA, États-Unis: O'Reilly Media.

Date, C. J. & Darwen, H., 1997. *A Guide to the SQL Standard: A user's guide to the standard database language SQL*. 4e éd. Boston, MA, États-Unis: Addison-Wesley.

Date, C. J. & Darwen, H., 2006. *Databases, Types, and the Relational Model: The Third Manifesto*. 3e éd. Boston, MA, États-Unis: Addison-Wesley.

Ecma International, 2006. *Standard ECMA-367 - Eiffel: Analysis, Design and Programming Language*. 2e éd. Genève, Suisse: Ecma International.

Meyer, B., 1986. Genericity versus Inheritance. Dans: *OOPSLA '86: Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*. New York, NY, États-Unis: ACM, pp. 391-405.

Meyer, B., 1988. Harnessing Multiple Inheritance. *Journal of Object-Oriented Programming,* nov., 1(4), pp. 48-51.

Meyer, B., 1992. Applying "Design by Contract". *Computer (IEEE),* oct., 25(10), pp. 40-51.

Mitchell, J. C., 2002. *Concepts in Programming Languages*. New York, NY, États-Unis: Cambridge University Press.

Nagy, L. P., 2007. *Type Inference, Type Improvement, and Type Simplification in a Language with User-Defined Polymorphic Relational Operators,* s.l.: Thèse de doctorat, Florida Institute of Technology.

Neward, T., 2006. *The Vietnam of Computer Science*. [En ligne]
Available at: http://blogs.tedneward.com/post/the-vietnam-of-computer-science/

Pierce, B. C., 2002. *Types and Programming Languages*. Cambridge, MA, États-Unis: The MIT Press.

Taivalsaari, A., 1996. On the Notion of Inheritance. *ACM Computing Surveys,* sept., 28(3), pp. 438-479.